Lecture 21: Program Repair

17-355/17-665/17-819: Program Analysis

Rohan Padhye

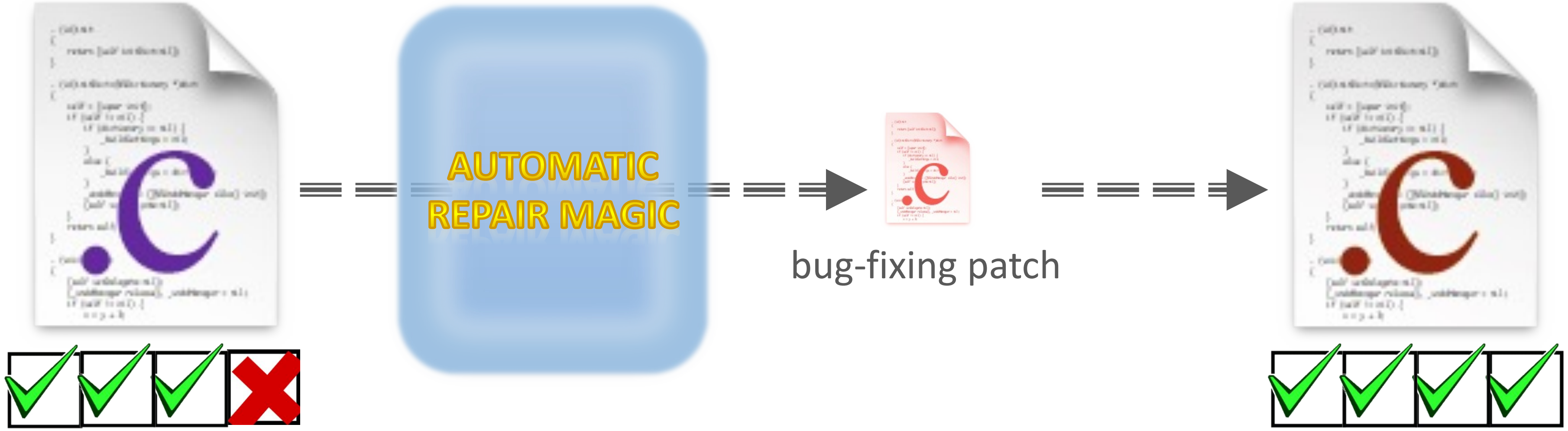
April 12, 2022

* Slides borrowed from Claire Le Goues

We’ve spent a lot of time on *finding* bugs

- What about *fixing* them?
- Problem: Given a program and an indication of a bug, find a patch for that program to fix that bug.
  - Both static and dynamic techniques have been used to “indicate” bugs.
  - The bulk of repair research is *dynamic*, or uses tests.
  - (We’ll talk about static briefly, and again later.)

Automatic Program Repair



Bug fixing: the 30000-foot view

1. Localize the bug.
   - And perform additional analysis
2. Create/combine fix possibilities into 1+ possible patches.
3. Validate candid

Fault localization

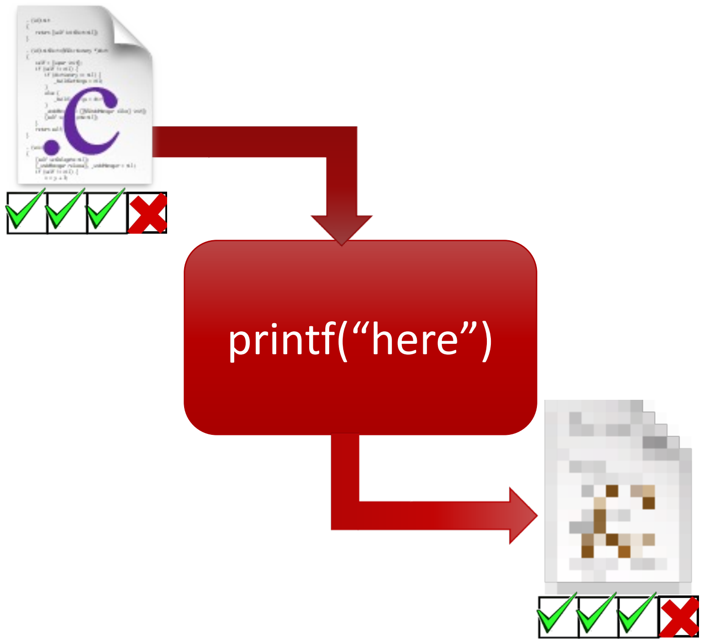
Tests.

Fault Localization

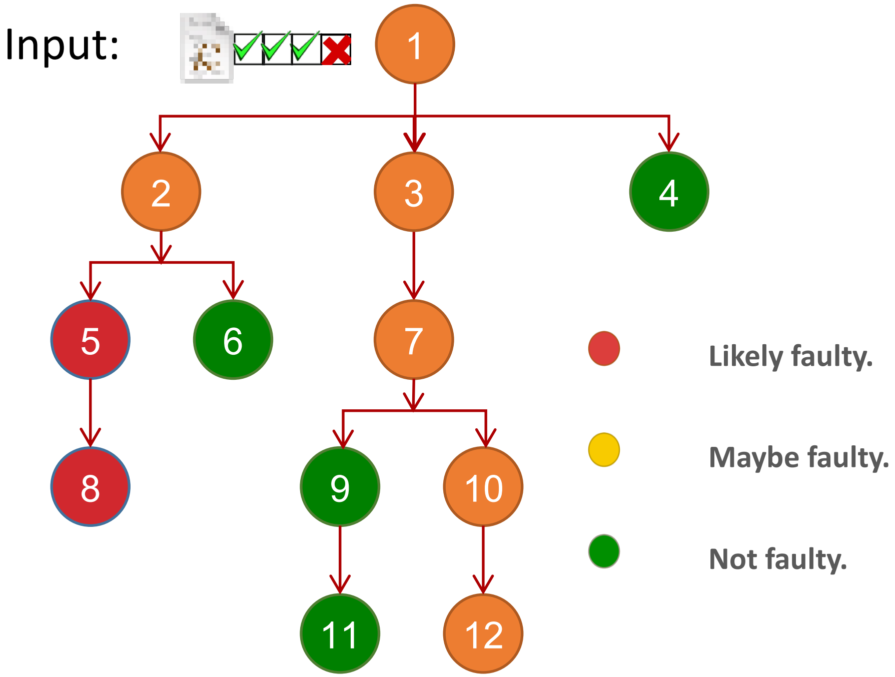
- Given: set of test cases, some of which fail
- To find: part of the code that's causing the failure
  - (which needs to be fixed)
- How is this done manually?
  - Printf("here")

Spectrum-Based Fault Localization

Automatically ranks potentially buggy program pieces based on test case behavior.



printf("here")



Input:

🟥 Likely faulty.

🟨 Maybe faulty.

🟩 Not faulty.

GenProg: Repair with Evolutionary Computation

1. Localize the bug.
   - And perform additional analysis
2. Create/combine fix possibilities into 1+ possible patches.
3. Validate candidate patches.

”GenProg: A generic method for automatic software repair” by Le Goues et al. IEEE TSE (2011)

Localize to C statements

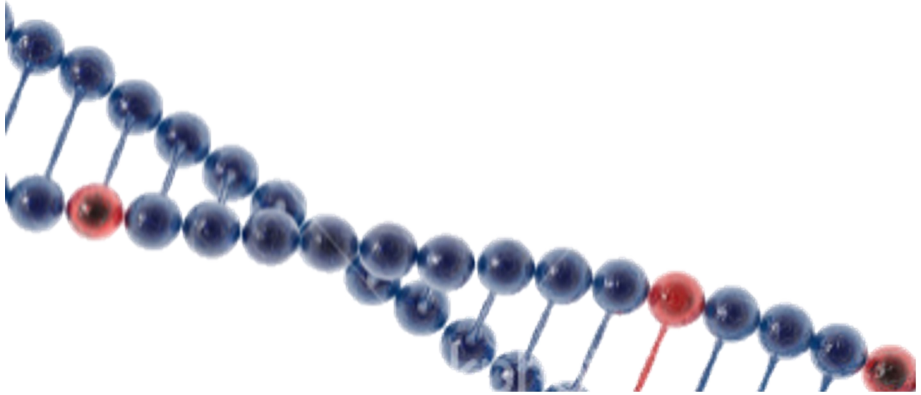
Genetic programming

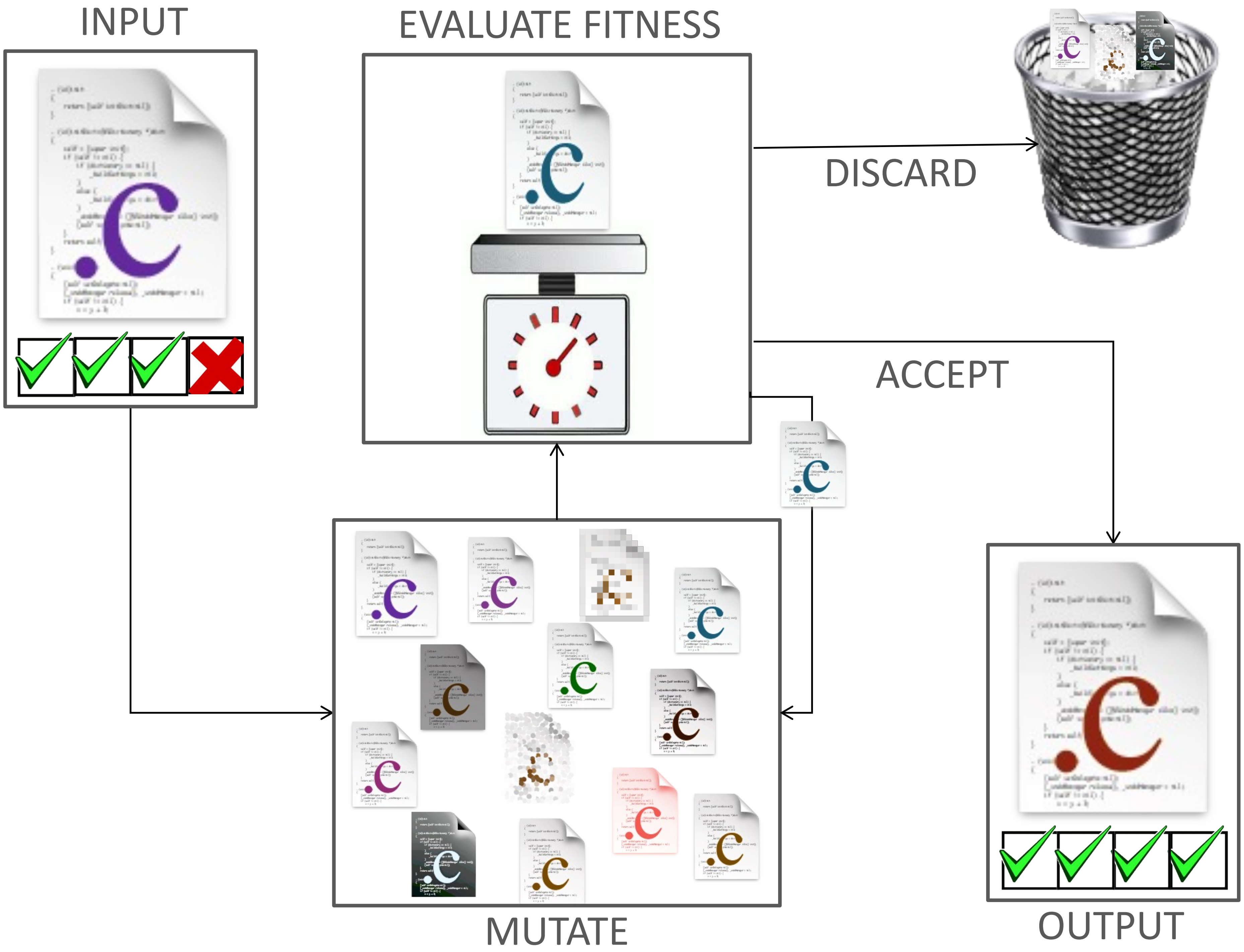
GenProg: Repair with Evolutionary Computation

Biased, random search for AST-level edits to a program that fixes a given bug without breaking any previously-passing tests.

Genetic Programming

The application of evolutionary or genetic algorithms to program source code.





INPUT

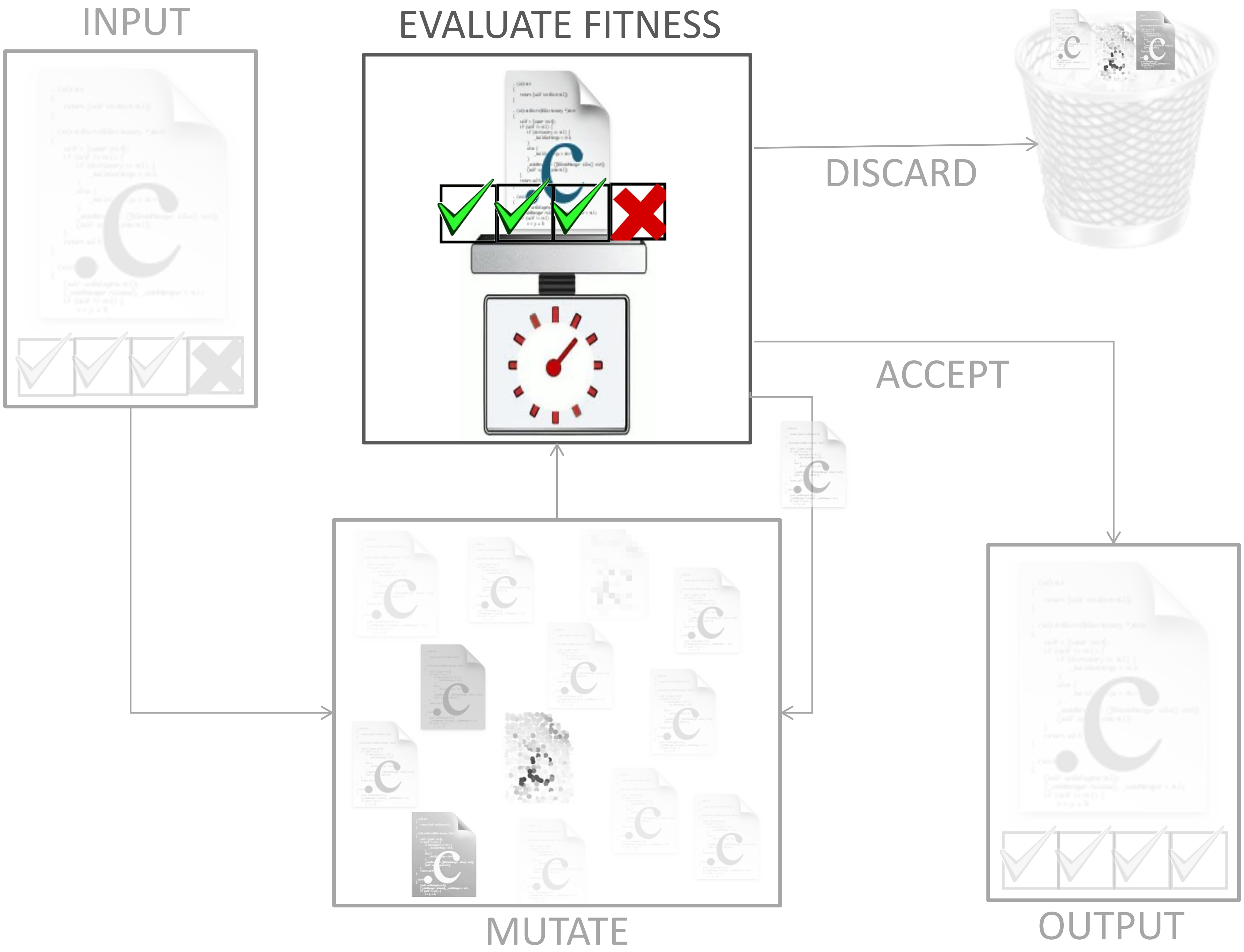
EVALUATE FITNESS

DISCARD

ACCEPT

MUTATE

OUTPUT



INPUT

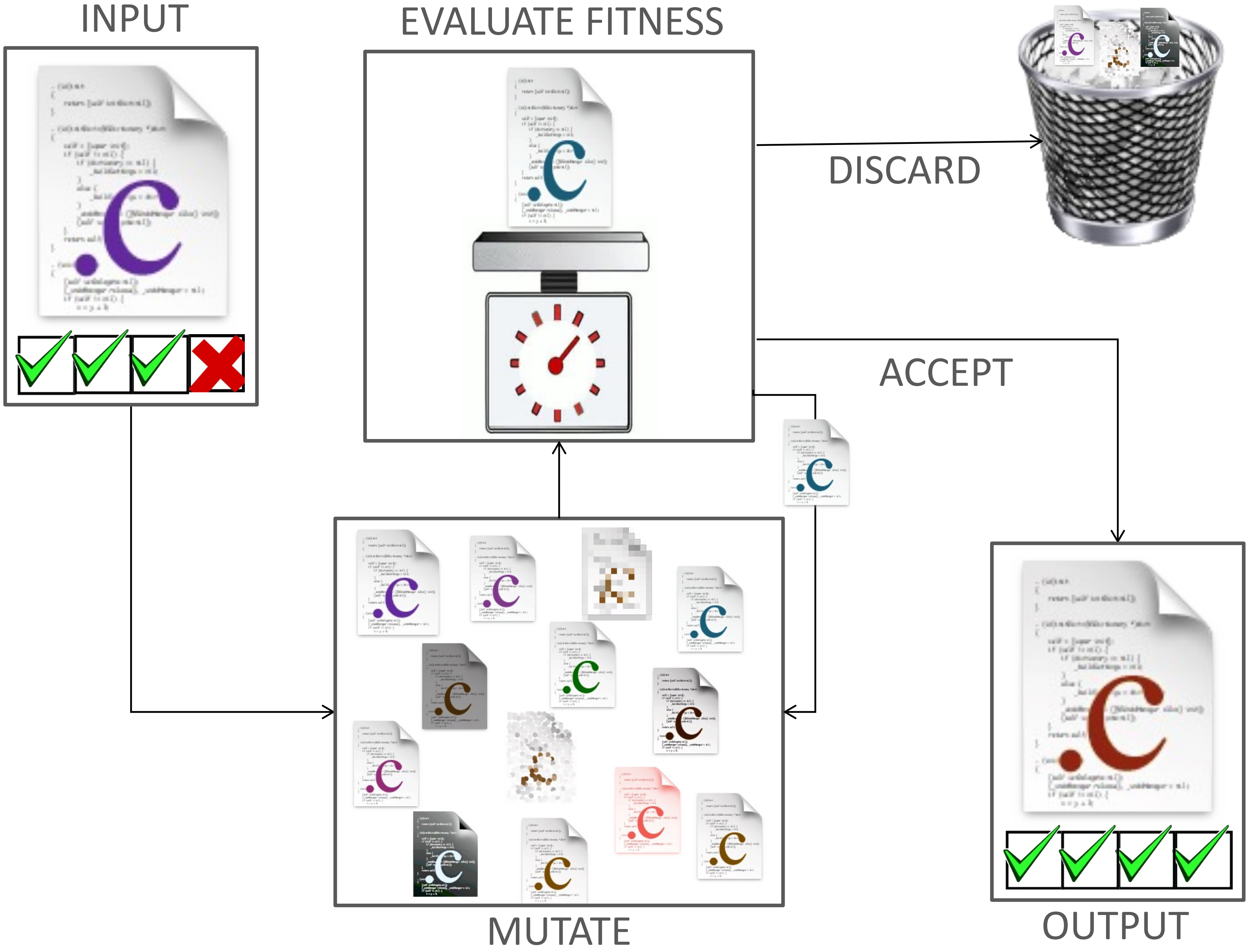
EVALUATE FITNESS

DISCARD

ACCEPT

MUTATE

OUTPUT



INPUT

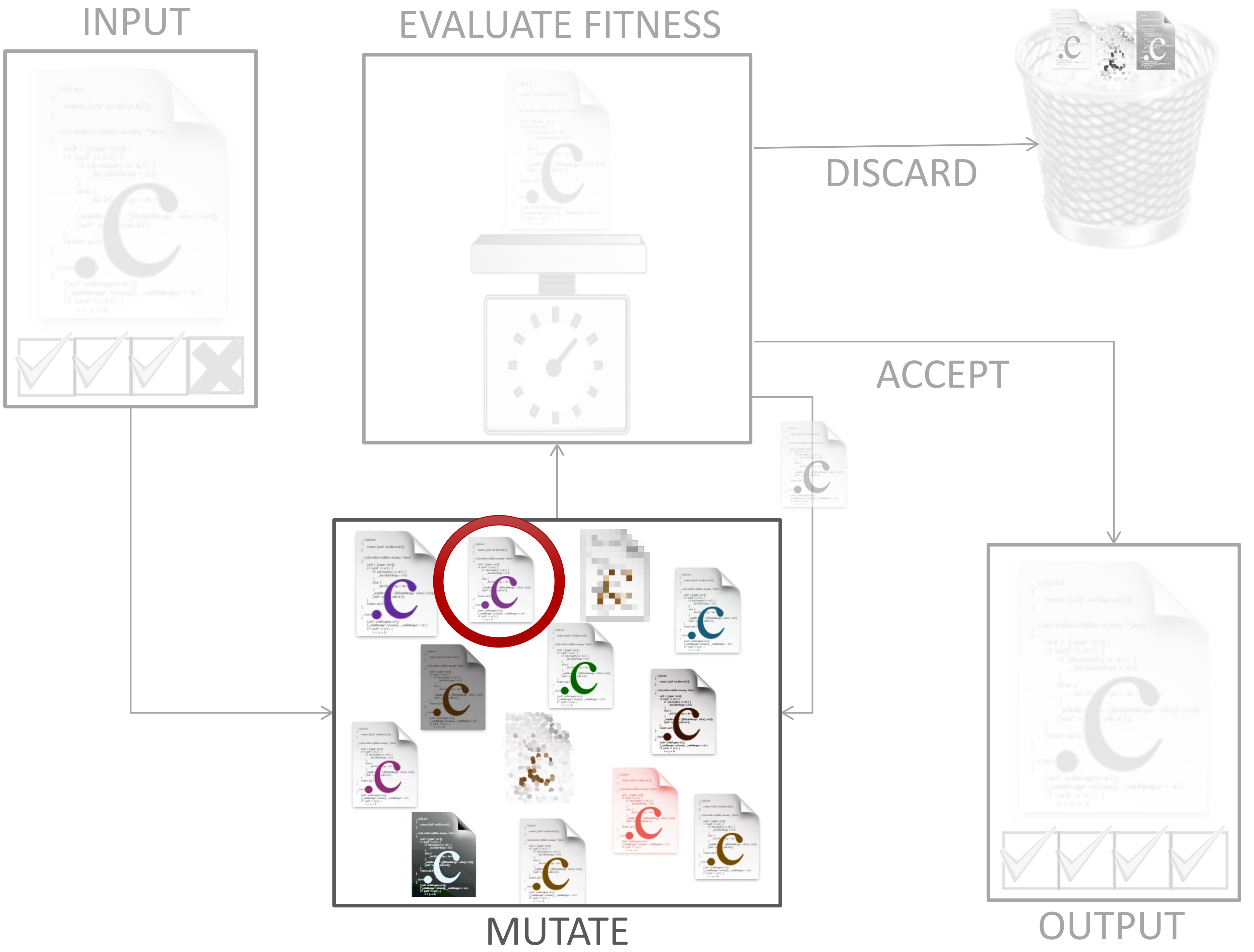
EVALUATE FITNESS

DISCARD

ACCEPT

MUTATE

OUTPUT


INPUT
EVALUATE FITNESS
DISCARD
ACCEPT
MUTATE
OUTPUT

An individual is a candidate patch/set of changes to the input program.

- A patch is a series of *statement-level* edits:
  - delete X
  - replace X with Y
  - insert Y after X.
- Replace/insert: pick Y from somewhere else in the program.
- To mutate an individual, add new random edits to a given (possibly empty) patch.
  - (Where? Right: fault localization!)

>

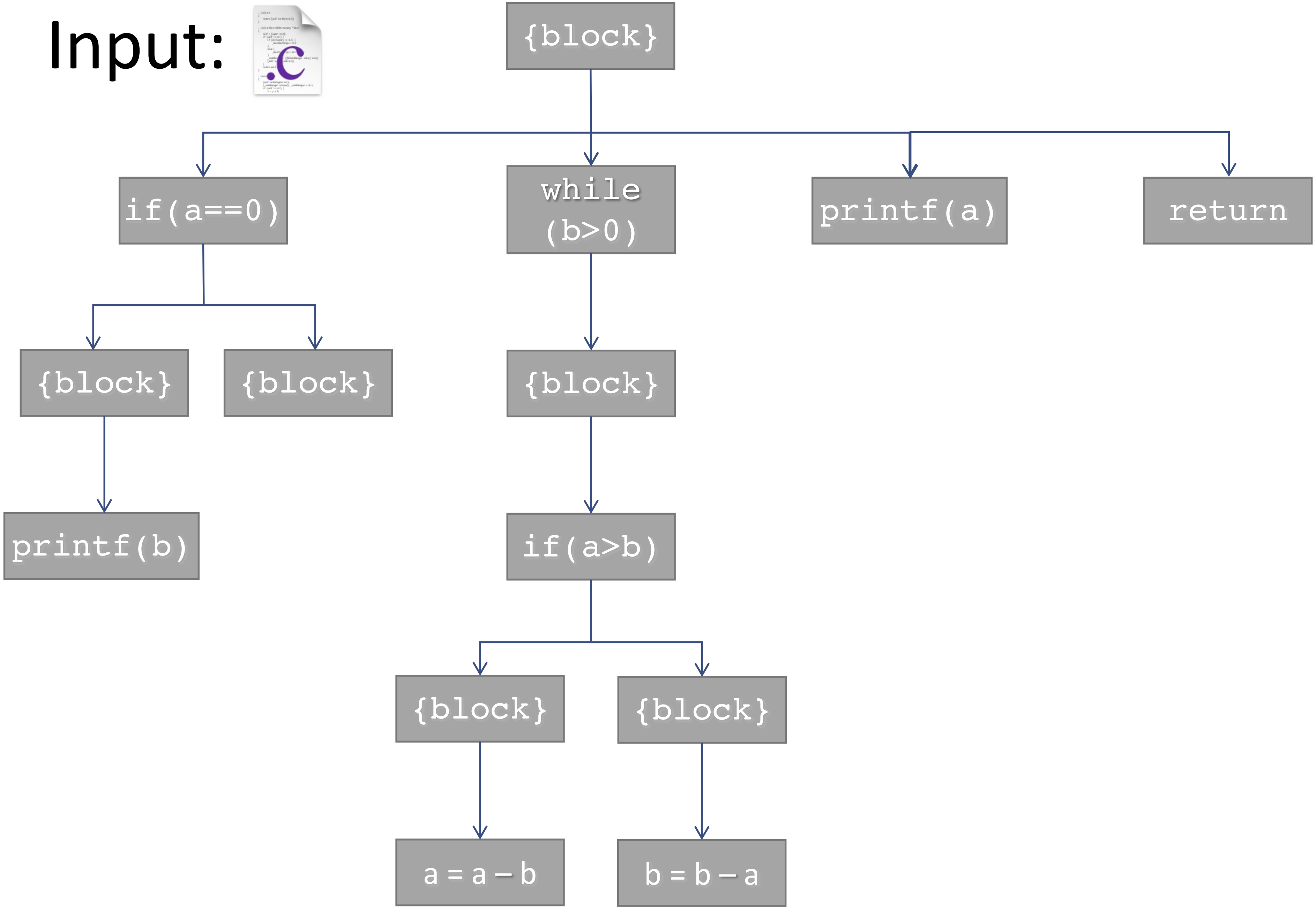
```
void gcd(int a, int b) {
  if (a == 0) {
    printf(“%d”, b);
  }
  while (b > 0) {
    if (a > b)
      a = a – b;
    else
      b = b – a;
  }
  printf(“%d”, a);
  return;
}
```

```
> gcd(4,2)
> 2
>
> gcd(1071,1029)
> 21
>
> gcd(0,55)
> 55
```

(looping forever)

```
void gcd(int a, int b) {
  if (a == 0) {
    printf(“%d”, b);
  }
  while (b > 0) {
    if (a > b)
      a = a – b;
    else
      b = b – a;
  }
  printf(“%d”, a);
  return;
}
```

Input:



{block}

if(a==0)

while (b>0)

printf(a)

return

{block}

{block}

printf(b)

{block}

if(a>b)

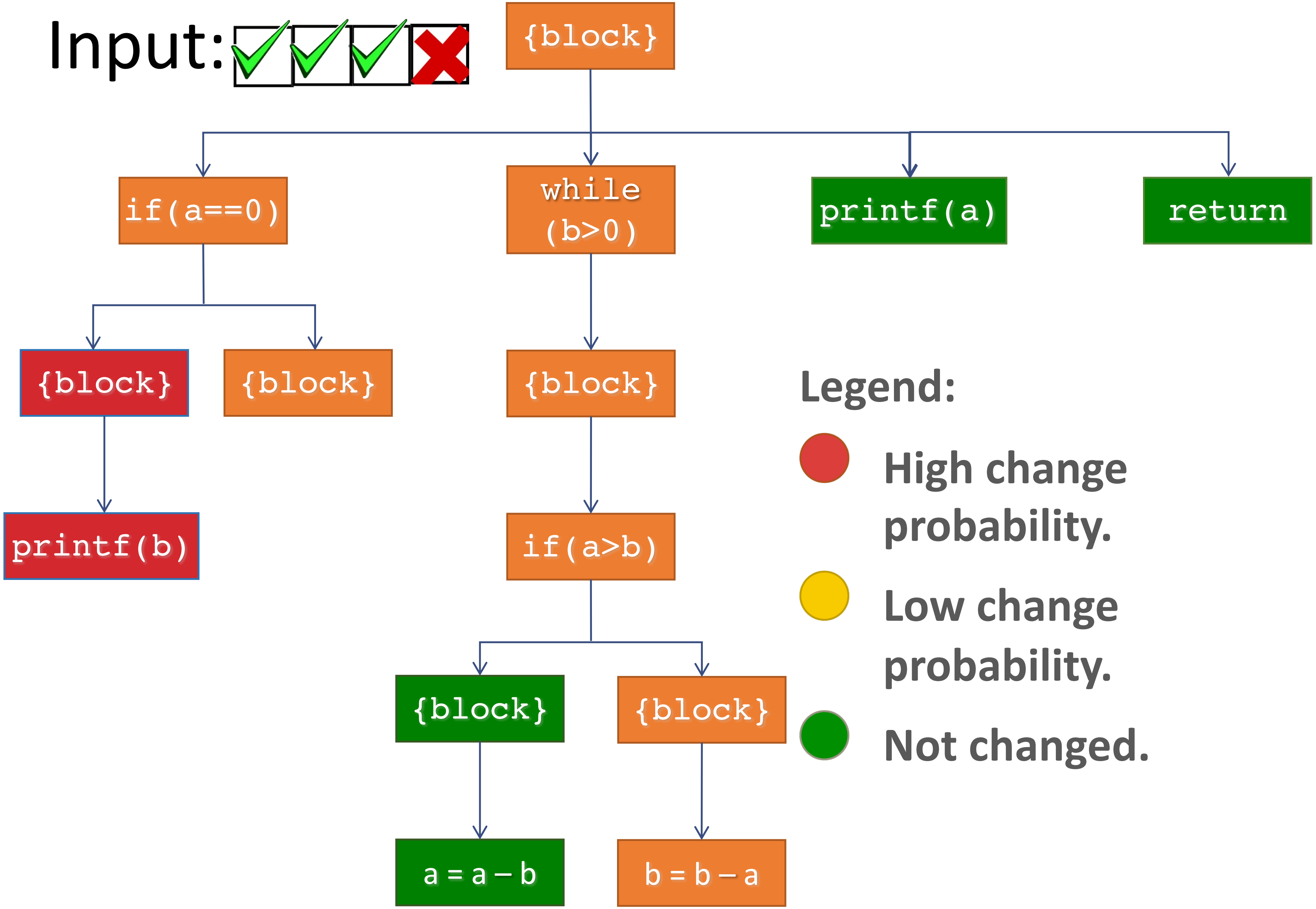
{block}

{block}

a = a – b

b = b – a

Input: ✓ ✓ ✓ ✗



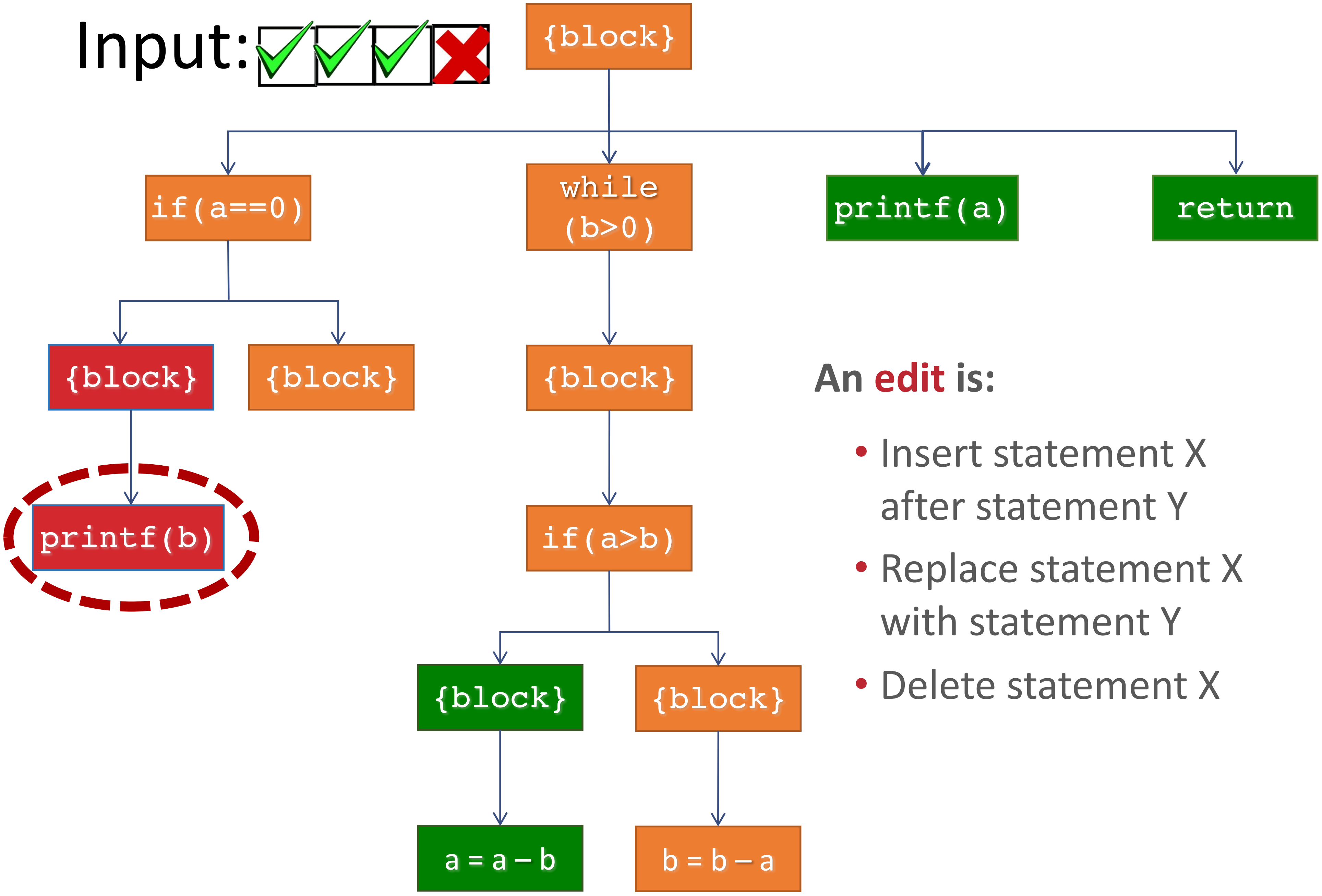
Legend:

🟥 High change probability.

🟨 Low change probability.

🟩 Not changed.

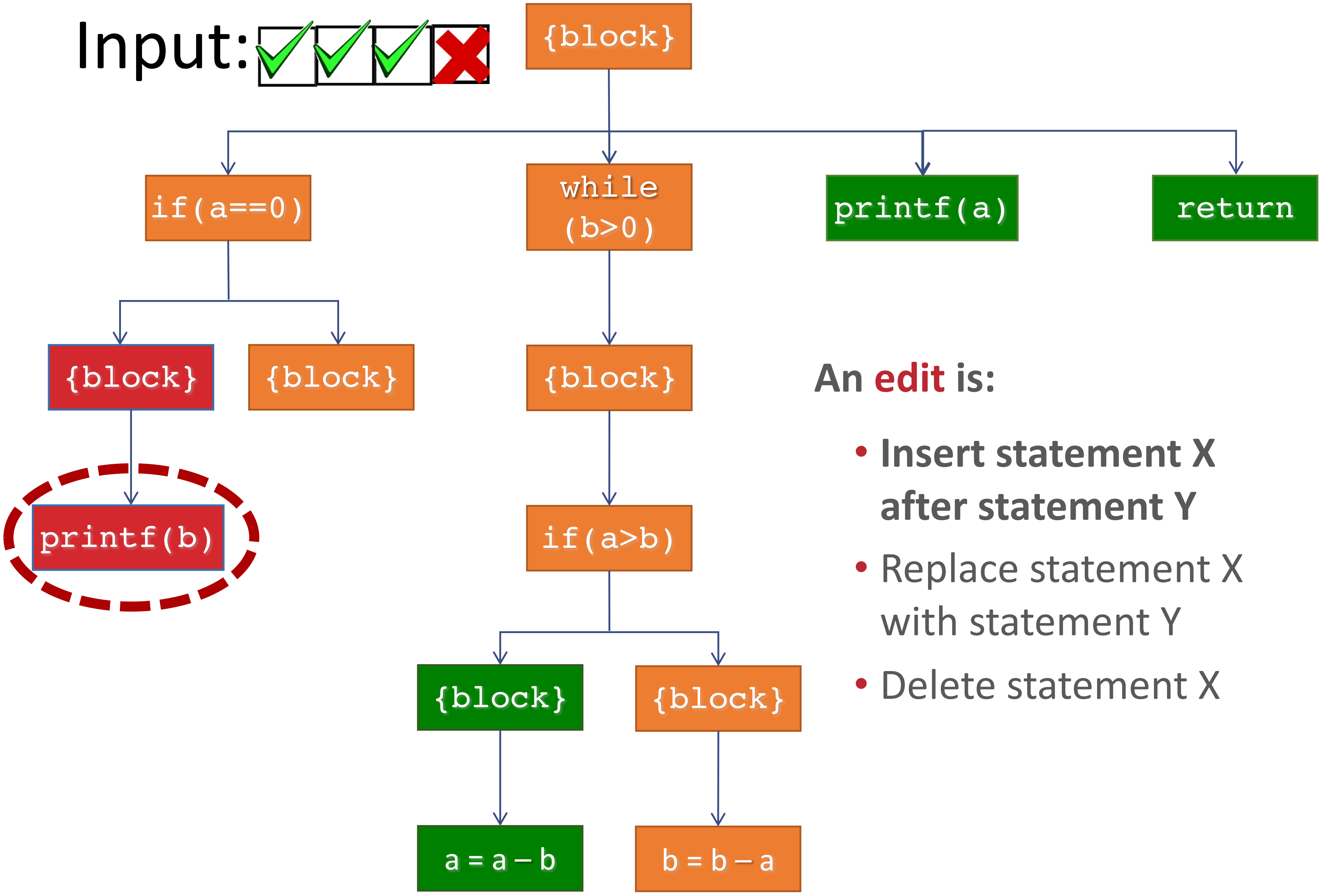
Input: ☑☑☑☒



An edit is:

- Insert statement X after statement Y
- Replace statement X with statement Y
- Delete statement X

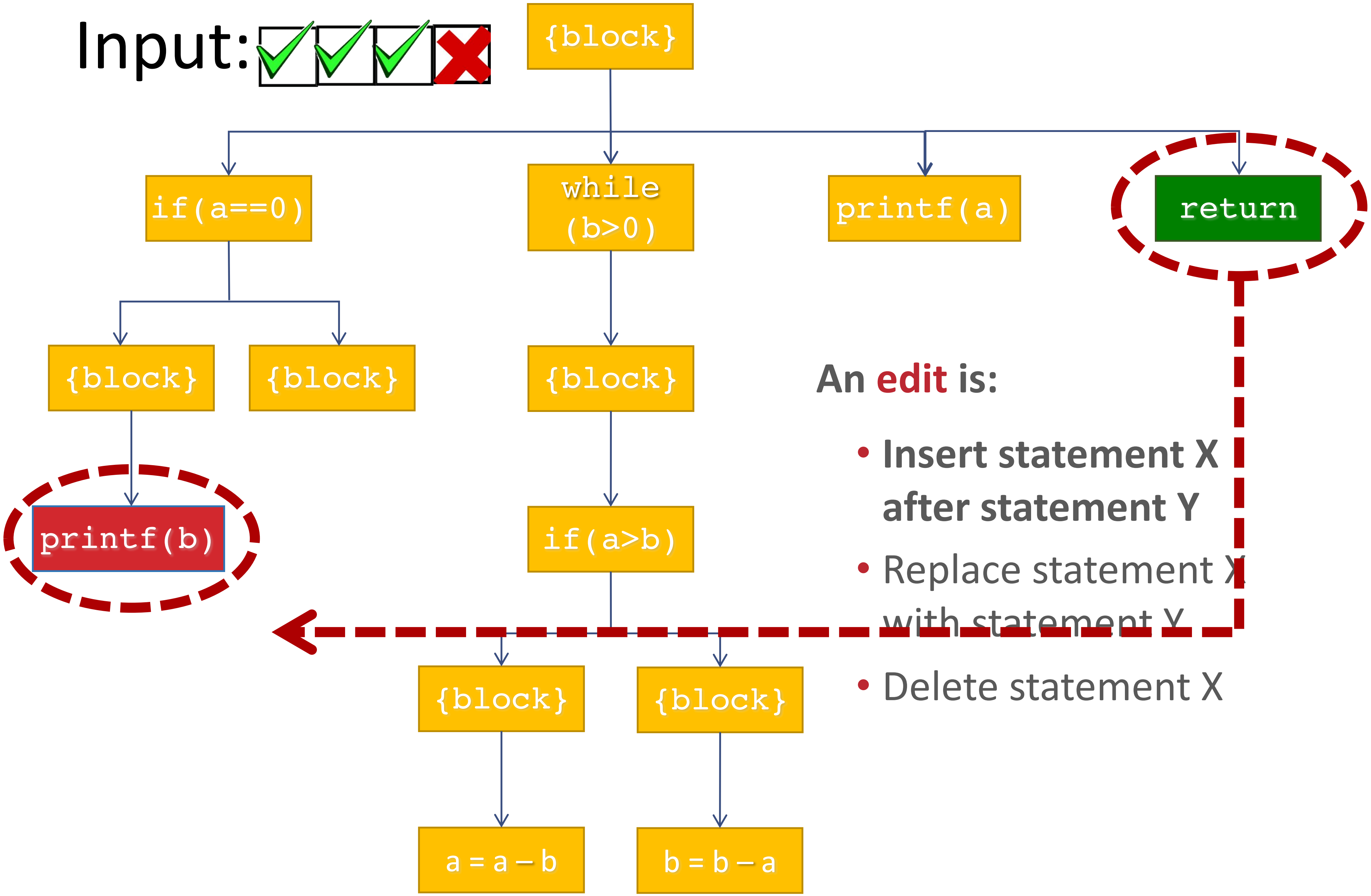
Input: ✓✓✓✗



An **edit** is:

- **Insert statement X after statement Y**
- Replace statement X with statement Y
- Delete statement X

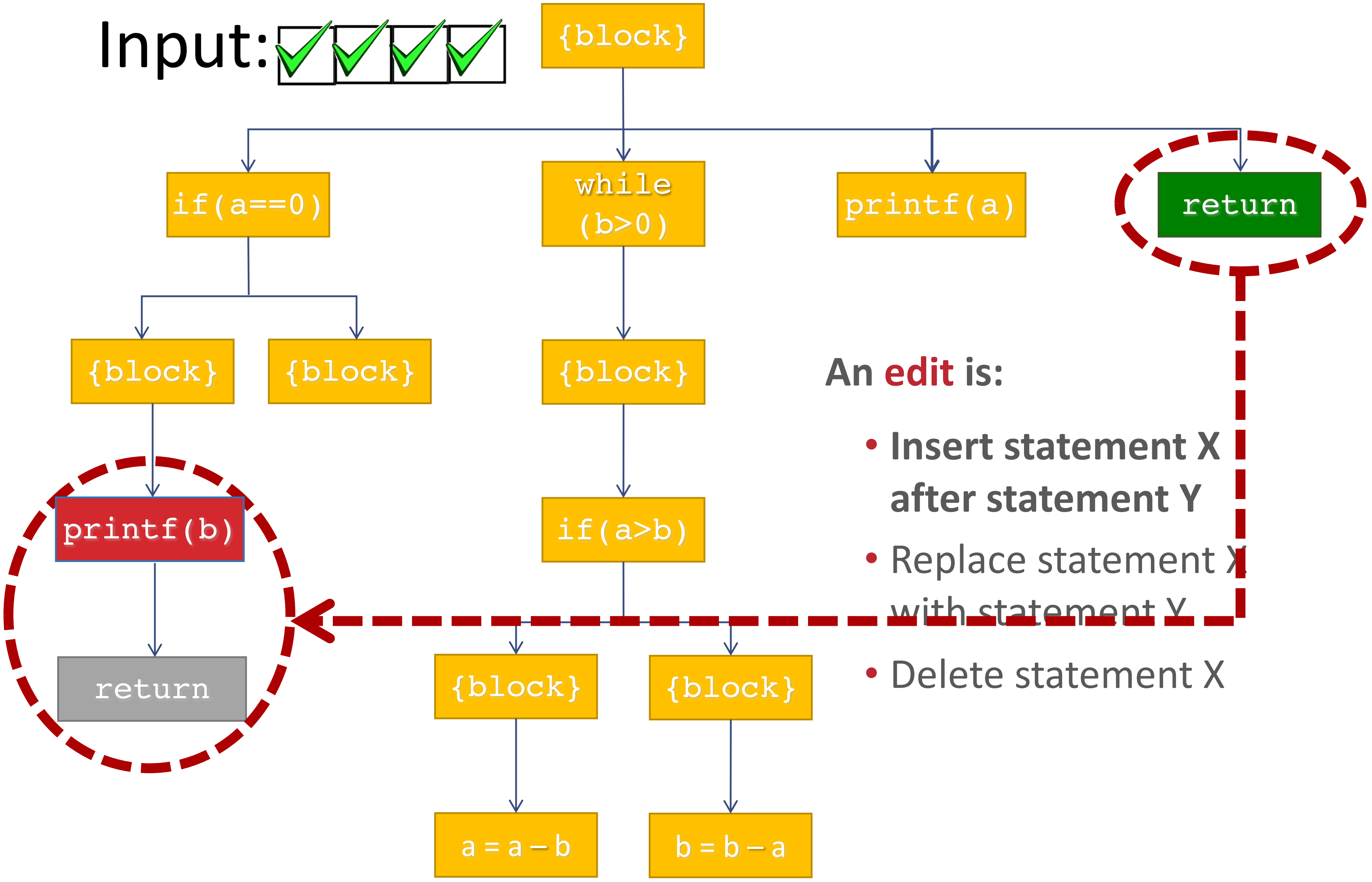
Input: ✓✓✓✗



An **edit** is:

- **Insert statement X after statement Y**
- Replace statement X with statement Y
- Delete statement X

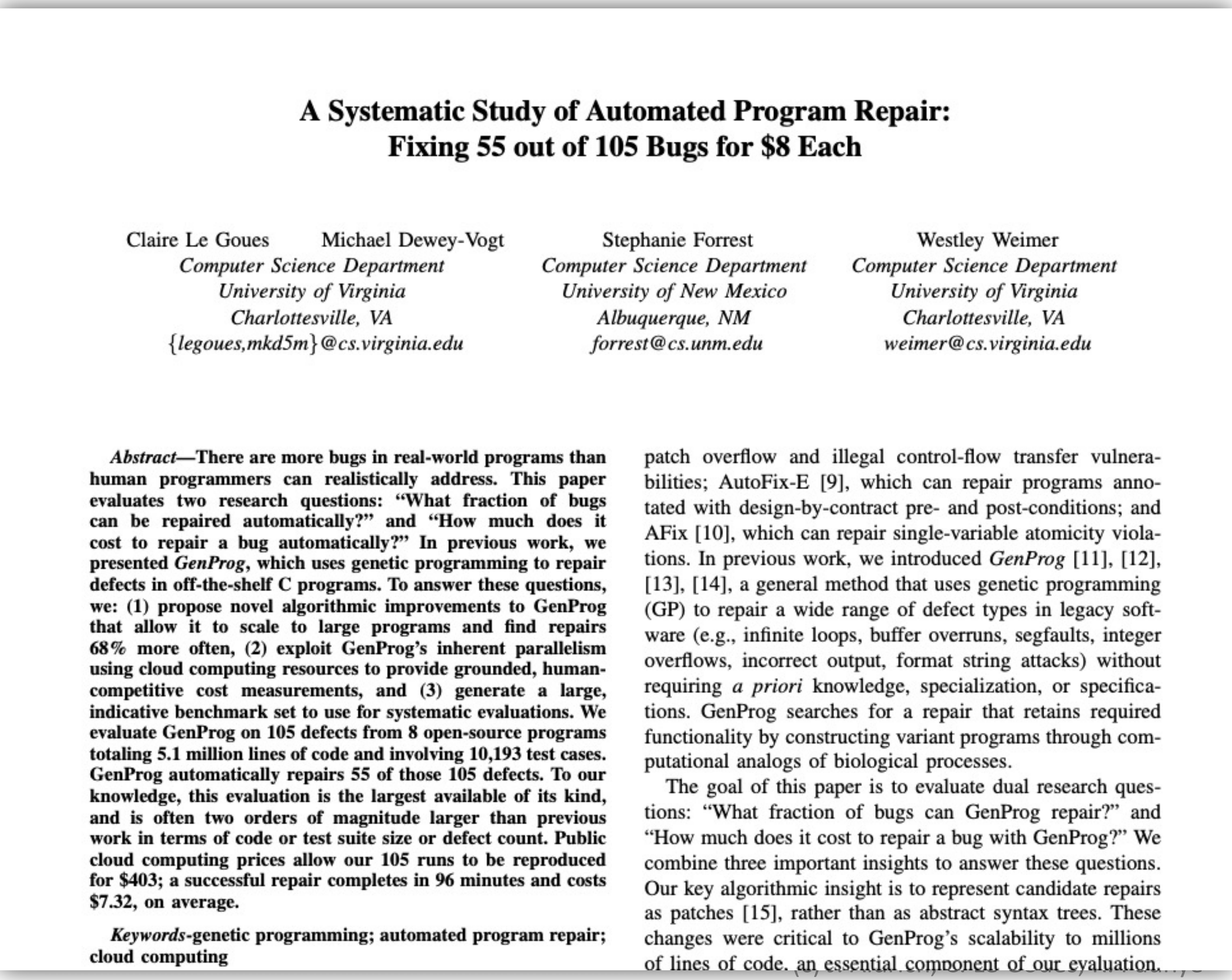
Input: ☑☑☑☑



An **edit** is:

- **Insert statement X after statement Y**
- Replace statement X with statement Y
- Delete statement X

Wait.. Isn't this expensive?



A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each

Claire Le Goues Michael Dewey-Vogt
Computer Science Department
University of Virginia
Charlottesville, VA
{legoues,mkd5m}@cs.virginia.edu

Stephanie Forrest
Computer Science Department
University of New Mexico
Albuquerque, NM
forrest@cs.unm.edu

Westley Weimer
Computer Science Department
University of Virginia
Charlottesville, VA
weimer@cs.virginia.edu

***Abstract*—There are more bugs in real-world programs than human programmers can realistically address. This paper evaluates two research questions: "What fraction of bugs can be repaired automatically?" and "How much does it cost to repair a bug automatically?" In previous work, we presented *GenProg*, which uses genetic programming to repair defects in off-the-shelf C programs. To answer these questions, we: (1) propose novel algorithmic improvements to GenProg that allow it to scale to large programs and find repairs 68% more often, (2) exploit GenProg's inherent parallelism using cloud computing resources to provide grounded, human-competitive cost measurements, and (3) generate a large, indicative benchmark set to use for systematic evaluations. We evaluate GenProg on 105 defects from 8 open-source programs totaling 5.1 million lines of code and involving 10,193 test cases. GenProg automatically repairs 55 of those 105 defects. To our knowledge, this evaluation is the largest available of its kind, and is often two orders of magnitude larger than previous work in terms of code or test suite size or defect count. Public cloud computing prices allow our 105 runs to be reproduced for $403; a successful repair completes in 96 minutes and costs $7.32, on average.**

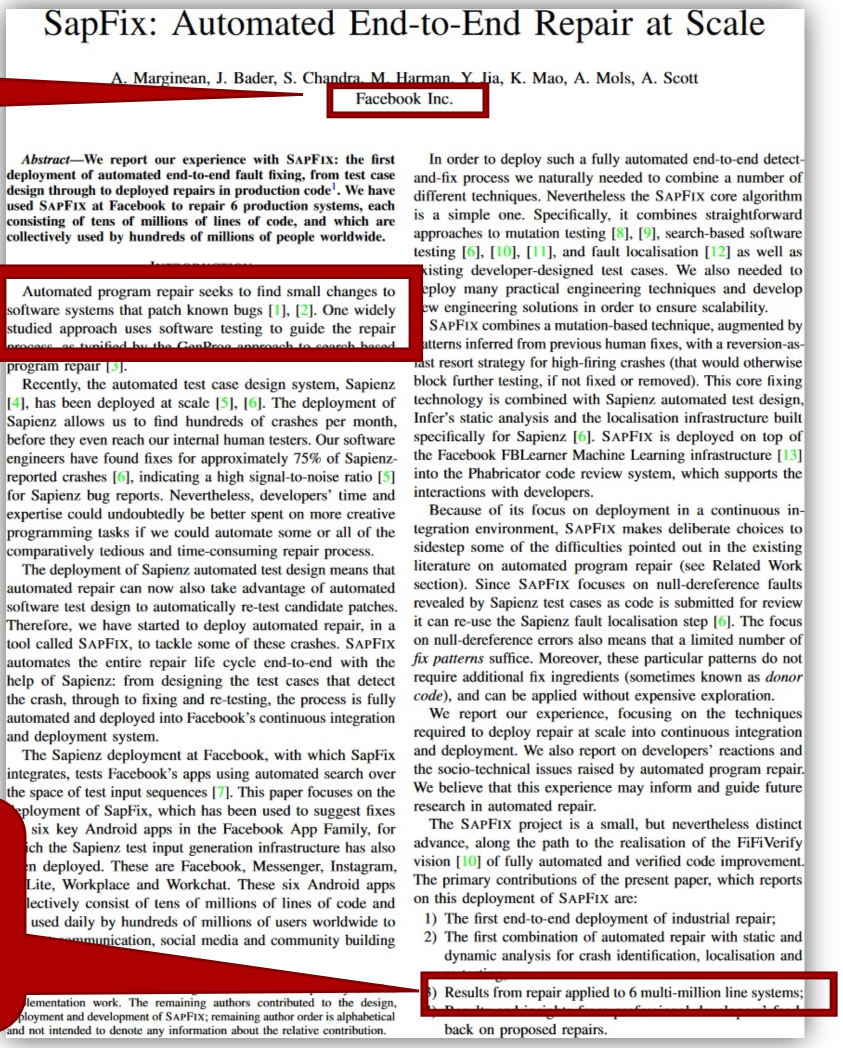
***Keywords*-genetic programming; automated program repair; cloud computing**

patch overflow and illegal control-flow transfer vulnerabilities; AutoFix-E [9], which can repair programs annotated with design-by-contract pre- and post-conditions; and AFix [10], which can repair single-variable atomicity violations. In previous work, we introduced *GenProg* [11], [12], [13], [14], a general method that uses genetic programming (GP) to repair a wide range of defect types in legacy software (e.g., infinite loops, buffer overruns, segfaults, integer overflows, incorrect output, format string attacks) without requiring *a priori* knowledge, specialization, or specifications. GenProg searches for a repair that retains required functionality by constructing variant programs through computational analogs of biological processes.

The goal of this paper is to evaluate dual research questions: "What fraction of bugs can GenProg repair?" and "How much does it cost to repair a bug with GenProg?" We combine three important insights to answer these questions. Our key algorithmic insight is to represent candidate repairs as patches [15], rather than as abstract syntax trees. These changes were critical to GenProg's scalability to millions of lines of code, an essential component of our evaluation.

“Real world” applications

“Facebook, Inc”



SapFix: Automated End-to-End Repair at Scale

A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, A. Scott

Facebook Inc.

***Abstract*—We report our experience with SapFix: the first deployment of automated end-to-end fault fixing, from test case design through to deployed repairs in production code[1]. We have used SapFix at Facebook to repair 6 production systems, each consisting of tens of millions of lines of code, and which are collectively used by hundreds of millions of people worldwide.**

Automated program repair seeks to find small changes to software systems that patch known bugs [1], [2]. One widely studied approach uses software testing to guide the repair process, as typified by the GenProg approach to search based program repair [3].

Recently, the automated test case design system, Sapienz [4], has been deployed at scale [5], [6]. The deployment of Sapienz allows us to find hundreds of crashes per month, before they even reach our internal human testers. Our software engineers have found fixes for approximately 75% of Sapienz-reported crashes [6], indicating a high signal-to-noise ratio [5] for Sapienz bug reports. Nevertheless, developers’ time and expertise could undoubtedly be better spent on more creative programming tasks if we could automate some or all of the comparatively tedious and time-consuming repair process.

The deployment of Sapienz automated test design means that automated repair can now also take advantage of automated software test design to automatically re-test candidate patches. Therefore, we have started to deploy automated repair, in a tool called SapFix, to tackle some of these crashes. SapFix automates the entire repair life cycle end-to-end with the help of Sapienz: from designing the test cases that detect the crash, through to fixing and re-testing, the process is fully automated and deployed into Facebook’s continuous integration and deployment system.

The Sapienz deployment at Facebook, with which SapFix integrates, tests Facebook’s apps using automated search over the space of test input sequences [7]. This paper focuses on the eployment of SapFix, which has been used to suggest fixes six key Android apps in the Facebook App Family, for ich the Sapienz test input generation infrastructure has also n deployed. These are Facebook, Messenger, Instagram, Lite, Workplace and Workchat. These six Android apps lectively consist of tens of millions of lines of code and used daily by hundreds of millions of users worldwide to communication, social media and community building

In order to deploy such a fully automated end-to-end detect-and-fix process we naturally needed to combine a number of different techniques. Nevertheless the SapFix core algorithm is a simple one. Specifically, it combines straightforward approaches to mutation testing [8], [9], search-based software testing [6], [10], [11], and fault localisation [12] as well as xisting developer-designed test cases. We also needed to eploy many practical engineering techniques and develop ew engineering solutions in order to ensure scalability.

SapFix combines a mutation-based technique, augmented by atterns inferred from previous human fixes, with a reversion-as-last resort strategy for high-firing crashes (that would otherwise block further testing, if not fixed or removed). This core fixing technology is combined with Sapienz automated test design, Infer’s static analysis and the localisation infrastructure built specifically for Sapienz [6]. SapFix is deployed on top of the Facebook FBLearner Machine Learning infrastructure [13] into the Phabricator code review system, which supports the interactions with developers.

Because of its focus on deployment in a continuous integration environment, SapFix makes deliberate choices to sidestep some of the difficulties pointed out in the existing literature on automated program repair (see Related Work section). Since SapFix focuses on null-dereference faults revealed by Sapienz test cases as code is submitted for review it can re-use the Sapienz fault localisation step [6]. The focus on null-dereference errors also means that a limited number of *fix patterns* suffice. Moreover, these particular patterns do not require additional fix ingredients (sometimes known as *donor code*), and can be applied without expensive exploration.

We report our experience, focusing on the techniques required to deploy repair at scale into continuous integration and deployment. We also report on developers’ reactions and the socio-technical issues raised by automated program repair. We believe that this experience may inform and guide future research in automated repair.

The SapFix project is a small, but nevertheless distinct advance, along the path to the realisation of the FiFiVerify vision [10] of fully automated and verified code improvement. The primary contributions of the present paper, which reports on this deployment of SapFix are:

1) The first end-to-end deployment of industrial repair;
2) The first combination of automated repair with static and dynamic analysis for crash identification, localisation and re-testing;
3) Results from repair applied to 6 multi-million line systems;
4) Results and insights from professional developers’ feedback on proposed repairs.

lementation work. The remaining authors contributed to the design, ployment and development of SapFix; remaining author order is alphabetical and not intended to denote any information about the relative contribution.

“one widely-studied [repair] approach uses software testing to guide the repair process, as typified by GenProg.”

“Results from repair applied to 6 multi-million line systems.”

Can GenProg fix this?

- The checksum program should:
  - Take a single-line string as input.
  - Sum the integer codes of the characters, excluding the newline, modulo 64, plus the code for the space character.
- Buggy student assignment →

```
// …
while (next != '\n')
{
    scanf("%c", &next);
    sum += next;
}
sum = sum % 64 + 22;
       um;
```

Incorrectly includes the newline in the sum.

Wrong value: the ASCII value of space is 32, not 22.

Voila!

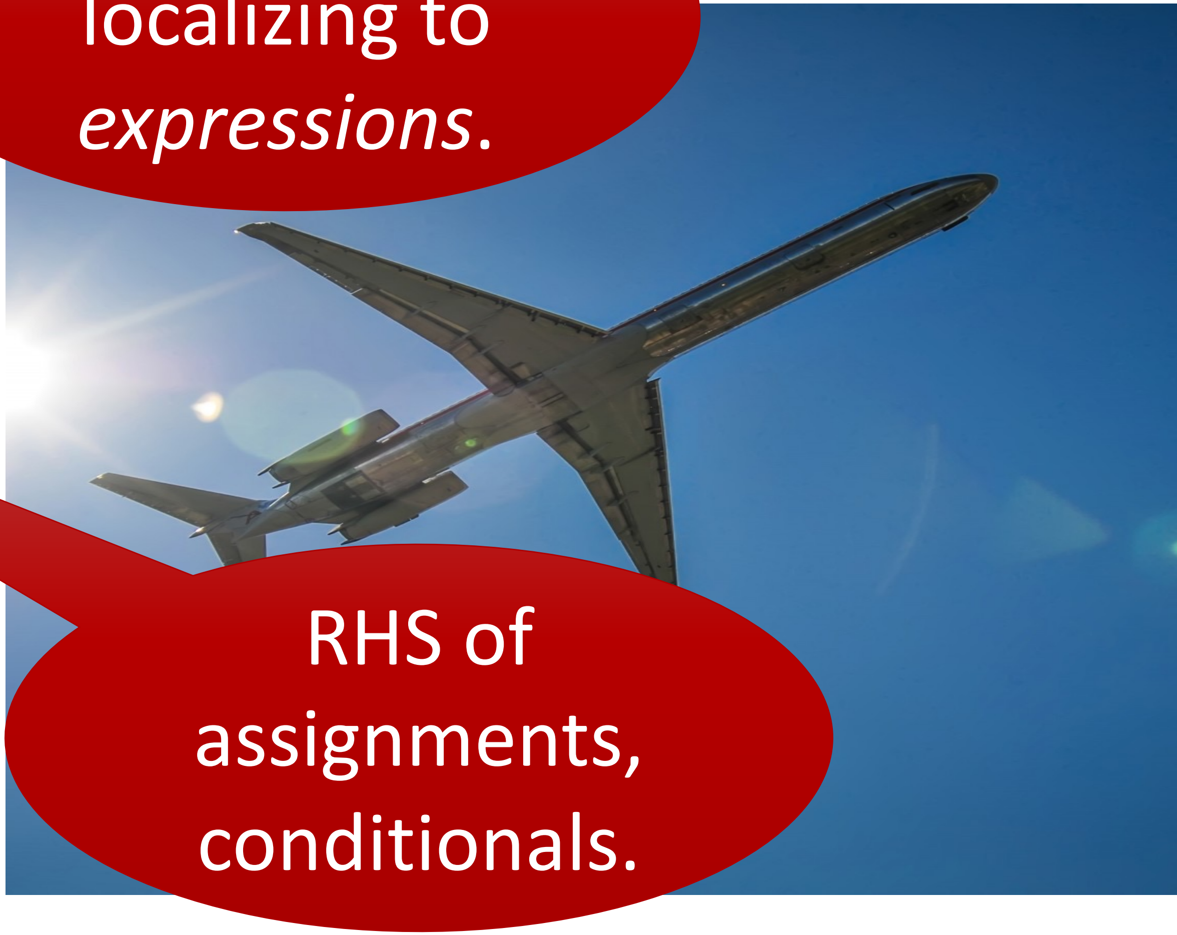
- The checksum program should:
  - Take a single-line string as input.
  - Sum the integer codes of the characters in the string, modulo 64, plus the code for the space character.
- GenProg fix with new representation →

```
// …
while (next != ‘\n’)
{
+  FIXME scanf(“%c”,
  &next);
     sum += next;
+ if (next == ‘\n’)
+       break;
}
sum = sum % 64 + 22;
+  sum += next;
return sum;
```

Semantics-based repair

1. Localize the bug.
   - And perform additional analysis
2. Create/combine fix possibilities into 1+ possible patches.
3. Validate candidate patch.

Same idea, but localizing to *expressions*.



RHS of assignments, conditionals.

”SemFix: Program Repair via Semantic Analysis” by Nguyen et al. ICSE 2013

“Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis” by Mechtaev et al. ICSE 2016

```c
int is_upward( int inhibit, int up_sep, int down_sep){
      int bias;
      if (inhibit)
          bias = down_sep;
      else  bias = up_sep ;
      if (bias > down_sep)
           return 1;
      else return 0;
}
```

(Slides by Abhik Roychoudhury)

```
int is_upward( int inhibit, int up_sep, int down_sep){
      int bias;
      if (inhibit)
          bias = down_sep;
      else  bias = up_sep ;
      if (bias > down_sep)
           return 1;
      else return 0;
}
```

inhibit	up_sep	down_sep	**Observed output**	**Expected Output**	**Result**
1	0	100	0	0	pass
1	***11***	***110***	***0***	***1***	***fail***
0	100	50	1	1	pass
1	***-20***	***60***	***0***	***1***	***fail***
0	0	10	0	0	pass

```
int is_upward( int inhibit, int up_sep, int down_sep){
      int bias;
      if (inhibit)
          bias = down_sep;
      else  bias = up_sep ;
      if (bias > down_sep)
           return 1;
      else return 0;
}
```

inhibit	up_sep	down_sep	**Observed output**	**Expected Output**	**Result**
1	0	100	0	0	pass
1	***11***	***110***	***0***	***1***	***fail***
0	100	50	1	1	pass
1	***-20***	***60***	***0***	***1***	***fail***
0	0	10	0	0	pass

```
int is_upward( int inhibit, int up_sep, int down_sep){
      int bias;
      if (inhibit)
          bias = down_sep; // bias= up_sep + 100
      else  bias = up_sep ;
      if (bias > down_sep)
           return 1;
      else return 0;
}
```

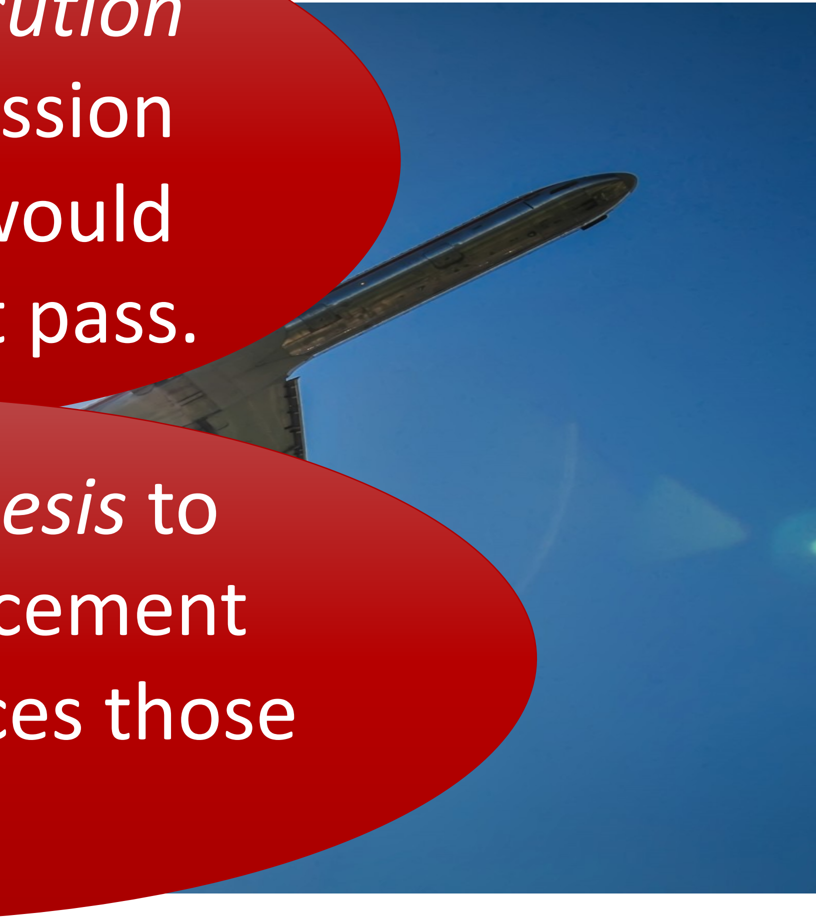
inhibit	up_sep	down_sep	**Observed output**	**Expected Output**	**Result**
1	0	100	0	0	pass
1	***11***	***110***	***0***	***1***	***fail***
0	100	50	1	1	pass
1	***-20***	***60***	***0***	***1***	***fail***
0	0	10	0	0	pass

Angelix

1. Localize the bug.
   - And perform additional analysis
2. Create/combine fix possibilities into 1+ possible patches.
3. Validate candidate patc

Concolic execution to find expression values that would make the test pass.

Program synthesis to construct replacement code that produces those values.



An expression's *angelic value* is the value that would make a given test case pass.

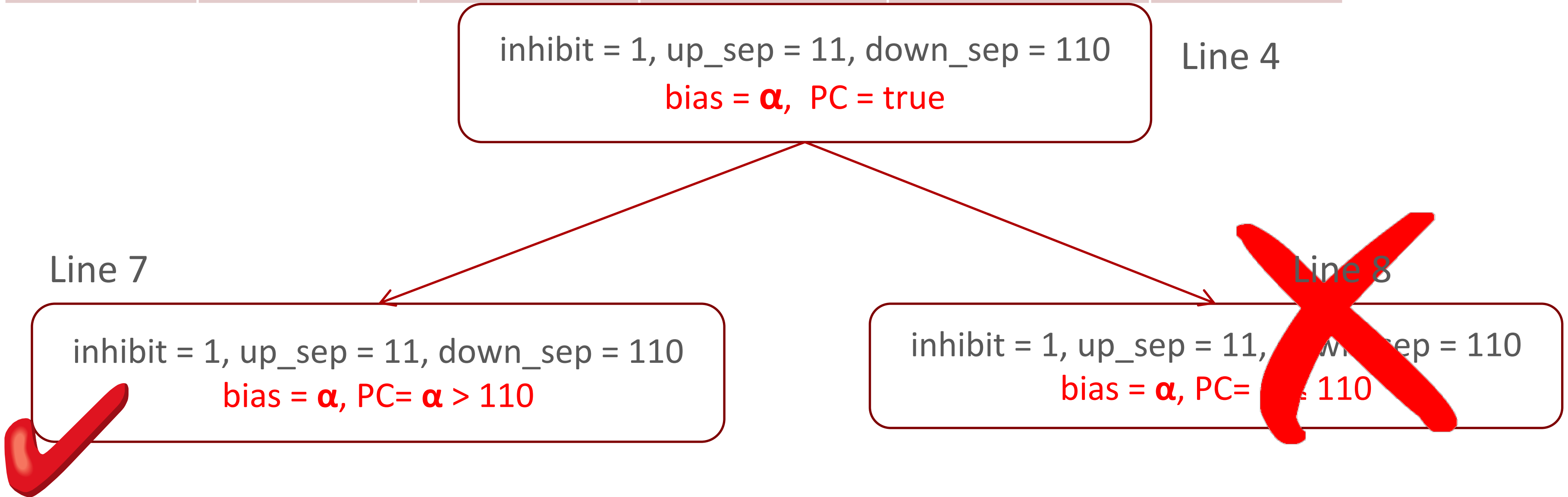
- This value is set "arbitrarily", by which we mean symbolically.
- You can *solve* for this value if you have:
  - the test case's expected input/output.
  - the path condition controlling its execution.
- Concolic execution (remember me?):
  - Start executing the test concretely, and then switch to symbolic execution when the angelic value starts to matter.

```
int is_upward( int inhibit, int up_sep, int down_sep){
        int bias;
        if (inhibit)
            bias = down_sep;
        else  bias = up_sep ;
        if (bias > down_sep)
             return 1;
        else return 0;
}
```

inhibit	up_sep	down_sep	**Observed output**	**Expected Output**	**Result**
1	0	100	0	0	pass
1	***11***	***110***	***0***	***1***	***fail***
0	100	50	1	1	pass
1	***-20***	***60***	***0***	***1***	***fail***
0	0	10	0	0	pass

```
1  int is_upward( int inhibit, int up_sep, int down_sep){
2        int bias;
3        if (inhibit)
4  ➡         bias = α;
5        else  bias = up_sep ;
6  ➡     if (bias > down_sep)
7  ➡          return 1;
8  ➡     else return 0;
9  }
```

inhibit	up_sep	down_sep	Observed output	Expected Output	**Result**
1	*11*	*110*	*0*	*1*	*fail*



inhibit = 1, up_sep = 11, down_sep = 110
bias = **α**, PC = true

Line 4

Line 7

inhibit = 1, up_sep = 11, down_sep = 110
bias = **α**, PC= **α** > 110

Line 8

inhibit = 1, up_sep = 11, down_sep = 110
bias = **α**, PC= **α** ≤ 110

```
int is_upward( int inhibit, int up_sep, int down_sep){
      int bias;
      if (inhibit)
          bias = α;
      else  bias = up_sep ;
      if (bias > down_sep)
           return 1;
      else return 0;
}
```

Exercise: Generate constraints for all other test cases

inhibit	up_sep	down_sep	**Observed output**	**Expected Output**	**Result**	**Constraint**
1	0	100	0	0	pass	
1	***11***	***110***	***0***	***1***	***fail***	f(1,11,110) > 110
0	100	50	1	1	pass	
1	***-20***	***60***	***0***	***1***	***fail***	
0	0	10	0	0	pass	

Collect all of the constraints!

- Accumulated constraints over all test cases:

$$f(1,11,110) > 110 \land f(1,0,100) \leq 100 \\ \land f(1,-20,60) > 60$$

- Use **oracle guided component-based program synthesis** to construct satisfying *f*:
  - *How does this work again?*
- Generated fix
  - `f(inhibit,up_sep,down_sep) = up_sep + 100` ✓

Heartbleed patch

```
if (hbtype == TLS1_HB_REQUEST
   && (payload + 18) < s->s3->rrec.length) {
   …
} else if (hbtype == TLS1_HB_RESPONSE) {
   …
}
return 0;
```

Generated patch

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
  return 0;
   …
if (hbtype == TLS1_HB_REQUEST) {
   …
} else if (hbtype == TLS1_HB_RESPONSE) {
   …
}
return 0;
```

Developer patch

https://upload.wikimedia.org/wikipedia/commons/thumb/d/dc/Heartbleed.svg/2000px-Heartbleed.svg.png

Challenges & Trade-Offs

1. Scalability

2. Expressibility of Repair

3. Patch Quality

Open problem: What is a high quality patch?

- Understandable?
- Doesn’t delete stuff?
  - Think: `assert(p)` ---> ~~`assert(p)`~~
- Addresses the cause, not the symptom…
  - Think: `++a[i]` ---> `try{++a[i]}catch(Exception e){}`
- Does the same thing the human did/would do?
  - But humans are often wrong! And how close does it have to be?